07 - Decrease and Conquer

[KOMS119602] & [KOMS120403]

Design and Analysis of Algorithm (2021/2022)

Dewi Sintiari

Prodi S1 Ilmu Komputer
Universitas Pendidikan Ganesha

Week 21-25 March 2022

Table of contents

Principal of Decrease-and-Conquer (1)

- This is similar to divide and conquer, except instead of partitioning a problem into multiple subproblems of smaller size, we use some technique to reduce our problem into a single problem that is smaller than the original.
- Some authors consider that the name "Divide-and-Conquer" should be used only when each problem may generate two or more subproblems. The name Decrease-and-Conquer has been proposed instead for the single-subproblem class. In the old literature, both of them are referred to as "Divide-and-Conquer".

Principal of Decrease-and-Conquer (2)

This approach is based on exploiting the relationship between a solution to a given instance of a problem and a solution to its smaller instance.

Implementation:

- Top-down approach: It always leads to the recursive implementation of the problem.
- Bottom-up approach: It is usually implemented in iterative way, starting with a solution to the smallest instance of the problem.

Principal of Decrease-and-Conquer (2)

Powering algorithm

Approach 1.

$$X^n = \begin{cases} 1 & \text{for } n = 0 \\ X^{n-1} \cdot X & \text{for } n > 0 \end{cases}$$

Approach 2.

$$X^n = \underbrace{X \cdot X \cdot X \cdot \cdots \cdot X}_{n \text{ times}}$$

Which approach is Top-down/Bottom-up?

of Decrease-and-Conquer

Three major variations of decrease-and-conquer

1. Decrease by a constant: the size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one.
2. Decrease by a constant factor: reducing a problem instance by the same constant factor on each iteration of the algorithm. In most applications, this constant factor is equal to two.
3. Decrease by a variable size: the size-reduction pattern varies from one iteration of an algorithm to another.

1. Decrease by a constant (1)

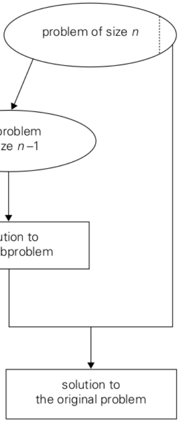


Figure: Decrease-(by one)-and-conquer technique.

1. Decrease by a constant (2)

Example: **Powering:** X^n

$$X^n = \begin{cases} 1 & \text{for } n = 0 \\ X^{n-1} \cdot X & \text{for } n > 0 \end{cases}$$

Here, X^n is computed by first decreasing the exponent by 1 (i.e. computing X^{n-1}).

1. Decrease by a constant (3)

Other examples:

- Selection sort
  This is a *hard split/easy join* algorithm by splitting the array into two sub-arrays: the first sub-array contains only *one element*, and the other sub-array contains $n-1$ elements.
- Insertion sort
  This is an *easy split/hard join* algorithm by splitting the array into two sub-arrays: the first sub-array contains only *one element*, and the other sub-array contains $n-1$ elements.

2. Decrease by a constant factor (1)

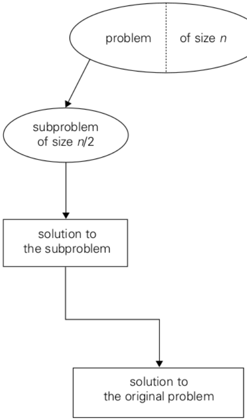


Figure: Decrease-(by half)-and-conquer technique.

2. Decrease by a constant factor (2)

Example 1: **Powering:** X^n

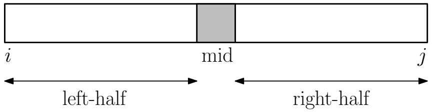
$$X^n = \begin{cases} 1 & \text{for } n = 0 \\ X^{n/2} \cdot X^{n/2} & \text{for } n > 0 \end{cases}$$

Here, X^n is computed by first decreasing the exponent by a half (i.e. computing $X^{n/2}$).

2. Decrease by a constant factor (3)

Example 2: Binary search (*already discussed in Week 05*)

Given an array A sorted (in ascending order), and a value X. We want to check if X is contained in A, and find the position of X in A.



- If mid $\neq X$, then we binary-search X only in the left-half or the right-half.
- So the instance size decreases by half.

2. Decrease by a constant factor (4)

Example 3: Fake-Coin Problem

Problem

Among n identical-looking coins, one is fake. With a balance scale, we can compare any two sets of coins. That is, by tipping to the left, to the right, or staying even, the balance scale will tell whether the sets weigh the same or which of the sets is heavier than the other but not by how much.

Task: *Design an efficient algorithm for detecting the fake coin.*

An easier version of the problem: *assume that the fake coin is known to be lighter than the genuine one*

2. Decrease by a constant factor (5)

Strategy:

1. Divide n coins into two piles of $\lfloor n/2 \rfloor$ coins each.
2. If the piles weigh the same, the coin put aside must be fake.
3. Otherwise, we can proceed in the same manner with the lighter pile, which must be the one with the fake coin.

Time complexity:

$W(n)$: the number of weighings needed by this algorithm in the worst case.

$$W(n) = \begin{cases} 0 & \text{for } n = 1 \\ W(\lfloor n/2 \rfloor) + 1 & \text{for } n > 1 \end{cases}$$

$W(n) = \mathcal{O}(\log n)$.

3. Decrease by a variable size (1)

The size-reduction pattern varies from one iteration of an algorithm to another.

Example: Euclids algorithm for computing the greatest common divisor. The algorithm is based on the property:

$$gcd(m, n) = gcd(n,\ m \bmod n)$$

Though the value of the second argument is always smaller on the right-hand side than on the left-hand side, it decreases neither by a constant nor by a constant factor.

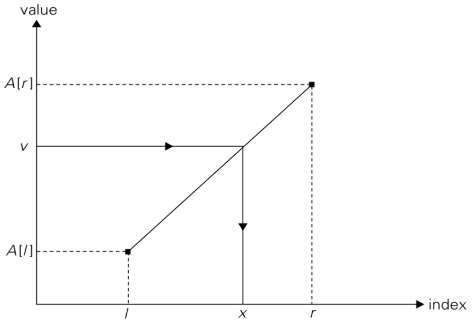
3. Decrease by a variable size (2): Interpolation search

Interpolation search

Problem: Given an array A sorted in ascending way, and a search key v. Check if v is in A, and find the position of v in A.

- Interpolation search takes into account the value of the search key in order to find the arrays element to be compared with the search key.
- The algorithm mimics the way we search for a name in a telephone book.
  - If we are searching for someone named "Brown", we open the book not in the middle but very close to the beginning, unlike our action when searching for someone named, say, "Smith".
- The algorithm assumes that the array values increase linearly, i.e., along the straight line through the points $(\ell, A[\ell])$ and $(r, A[r])$. (The accuracy of this assumption can influence the algorithms efficiency but not its correctness.)

Interpolation search (*cont.*)



$$\frac{v-A[\ell]}{A[r]-A[\ell]} = \frac{x-\ell}{r-\ell}$$

$$x = \ell + r - \ell \cdot \frac{v-A[\ell]}{A[r]-A[\ell]}$$

ℓ: the start index of A
r: the end index of A
v: the search key
x: the index of v

We do not use the constant $\frac{1}{2}$ as in Binary Search, but another more accurate constant "*c*", that can lead us closer to the searched item.

Interpolation search (*cont.*)

Recall the Binary Search algorithm

Algorithm 1 Binary search algorithm

```
procedure BinSearch($A, i, j, KEY$)
    if $i > j$ then
        return $-1$                                  ▷ Base case is reached but KEY is not found
    end if
    $m = \lfloor \frac{i+j}{2} \rfloor$                                       ▷ Choose the pivot
    if $KEY = A[m]$ then
        return $m$                                   ▷ KEY is found at index m
    else
        if $KEY < A[m]$ then                         ▷ The KEY is located on the Left sub-array
            return BinSearch($A, i, m-1, KEY$)       ▷ Rec-call left part
        else
            return BinSearch($A, m+1, j, KEY$)       ▷ Rec-call right part
        end if
    end if
end procedure
```

Interpolation search (*cont.*)

The Interpolation Search algorithm is similar to Binary Search, by replacing

$$\text{mid} \leftarrow (i + j) \ \textbf{div} \ 2$$

with

$$\text{mid} \leftarrow i + (j - i) * \left(\frac{v - A[i]}{A[j] - A[i]} \right)$$

(*In the previous figure,* $i = \ell$, $j = r$)

Interpolation search (*cont.*)

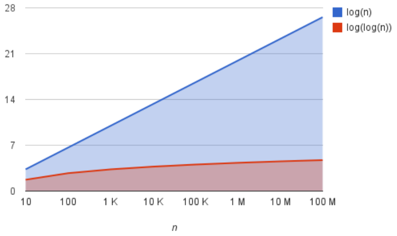
Algorithm 2 Interpolation search algorithm

```
1:  procedure InterpSearch($A, i, j, v$)
2:      if $i > j$ then
3:          return $-1$                          ▷ Base case is reached but v is not found
4:      end if
5:      $m = \lfloor i + (j - i) * \frac{v - A[i]}{A[j] - A[i]} \rfloor$                 ▷ Choose the pivot
6:      if $v = A[m]$ then
7:          return $m$                           ▷ v is found at index m
8:      else
9:          if $v < A[m]$ then                   ▷ v is located on the left sub-array
10:             return InterpSearch($A, i, m - 1, v$)        ▷ Rec-call left part
11:         else
12:             return InterpSearch($A, m + 1, j, v$)        ▷ Rec-call right part
13:         end if
14:     end if
15: end procedure
```

Interpolation search (*cont.*)

Time complexity of $\textsc{InterpSearch}$

- Best-case: $\mathcal{O}(1)$
- Worst-case: $\mathcal{O}(n)$, for any data distribution.
- Average-case: $\mathcal{O}(\log\log n)$, if the data in the array is *uniformly distributed*.



If n is 1 billion ($= 10^9$), $\log(\log(n)) \approx 5$ (computed in base 2), $\log(n) \approx 30$.

3. Decrease by a variable size (3): Computing median & Selection Problem

Selection problem is the problem of finding the kth smallest element in a list of n numbers. This number is called the kth order statistic.

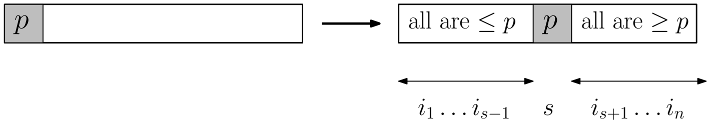
- If $k = 1$: finding the smallest element (*minimum*)
- If $k = n$: finding the largest element (*maximum*)
- If $k = \lceil \frac{n}{2} \rceil$: finding *median*

Obviously, we can find the kth smallest element in a list *by sorting the list first and then selecting the kth element in the output of a sorting algorithm*.

But, what if *we are not allowed to sort the array?*

Computing median & Selection Problem (*cont.*)

Use similar idea as "Partitioning" used in Quick Sort. But we do not always partition the search array in the middle.



- If $s = \lceil n/2 \rceil$, then the pivot p is the median
- if $s > \lceil n/2 \rceil$, then the median is in the left sub-array
- if $s < \lceil n/2 \rceil$, then the median is in the right sub-array

Computing median & Selection Problem (*cont.*)

Example of implementation of QUICKSELECT

Apply the *partition-based* algorithm to find the median

Here k = ⌈9/2⌉ = 5
So we want to find the element in the 5th position

$\boxed{x}$ **pivot**

0	1	2	3	4	5	6	7	8
$\boxed{\overset{s}{4}}$	$\overset{i}{1}$	10	8	7	12	9	2	15
$\boxed{4}$	$\overset{s}{1}$	$\overset{i}{10}$	8	7	12	9	2	15
$\boxed{4}$	$\overset{s}{1}$	10	8	7	12	9	$\overset{i}{2}$	15
$\boxed{4}$	1	$\overset{s}{2}$	8	7	12	9	$\overset{i}{10}$	15
$\boxed{4}$	1	$\overset{s}{2}$	8	7	12	9	10	$\overset{i}{15}$
2	1	$\boxed{4}$	8	7	12	9	10	15
			$\boxed{\overset{s}{8}}$	$\overset{i}{7}$	12	9	10	15
			$\boxed{8}$	$\overset{s}{7}$	$\overset{i}{12}$	9	10	15
			$\boxed{8}$	$\overset{s}{7}$	12	9	10	$\overset{i}{15}$
			7	$\boxed{8}$	12	9	10	15

Computing median & Selection Problem (*cont.*)

Algorithm 3 Lomuto partition

```
1:  procedure LOMUTO($A, \ell, r$)
2:      input: a sub-array $A[\ell..r]$ of array $A[0..n-1]$
3:      output: partition of $A[\ell..r]$ and the new position of the pivot
4:      $p \leftarrow A[\ell]$
5:      $s \leftarrow \ell$
6:      for $i \leftarrow \ell + 1$ to $r$ do
7:          if $A[i] < p$ then
8:              $s \leftarrow s + 1$
9:              swap($A[s], A[i]$)
10:         end if
11:     end for
12:     swap($A[\ell], A[s]$)
13:     return $s$
14: end procedure
```

Computing median & Selection Problem (*cont.*)

Algorithm 4 Quick Selection

```
1:  procedure QUICKSELECT($A, \ell, r, k$)
2:      input: a sub-array $A[\ell..r]$ of ordorable array $A[0..n-1]$, and integer $k$ with $1 \le k \le r - \ell + 1$
3:      output: the value of the $k$-th smallest element in $A[\ell..r]$
4:      $s \leftarrow$ LOMUTO($A, \ell, r$)
5:      if $s = k - 1$ then
6:          return $A[s]$
7:      else
8:          if $s > \ell + k - 1$ then
9:              QUICKSELECT($A, \ell, s - 1, k$)
10:         else
11:             QUICKSELECT($A, s + 1, r, k - 1 - s$)
12:         end if
13:     end if
14: end procedure
```

Computing median & Selection Problem (*cont.*)

Time complexity of Quickselect

Best-case

- Partitioning array of size n by LOMUTO always requires $n-1$ key comparisons.
- If it produces the split that produces the solution without recursive call, then:

$$TC_{\text{best}}(n) = n - 1 \in \Theta(n)$$

Worst-case

But, the algorithm can produce an extremely unbalanced partition of a given array, with one part being empty and the other containing $n-1$ elements.

In this case:

$$TC_{\text{worst}}(n) = (n-1) + (n-2) + \cdots + 1 = \frac{(n-1)n}{2} \in \Theta(n^2)$$